Using Supervised Learning Techniques to Predict Television Ratings

Computer Science Undergraduate Honors Thesis
University of Texas at Austin

Jackson Hassell

Supervised by:

Dr. Paul Navratil

Second Reader: Dr. Ray Mooney

Departmental Reader: Dr. Gordon Novak

Table of Contents

Abstract

How well a given TV show does is scored by a metric called “rating,” which denotes the percentage of households watching live TV at the time that are tuned into that particular show. To maximize ratings, being able to reliably predict them is necessary. For my thesis, in collaboration with Austin’s public-television station KLRU-TV, a variety of techniques were tested in order to discern the most accurate model for predicting the ratings of a television-show airing.

To accomplish this, I created nine regression models, each using a different algorithm that has been proven to work across many kinds of problems. These were a linear regression model, a k-nearest-neighbors model, a SVM model, a decision tree model, a bagging ensemble model, a gradient boosting ensemble model, two kinds of fully connected neural networks, or MLPs, and a recurrent neural network. I also created several feature sets, which included Nielsen, IMDb, and engineered features. Each model was tested across every combination of feature sets and exhaustively hyperparamatized to find what method produced the best results.

Most models did similarly well under at least one combination of hyperparameters and feature set, with the only exception being the linear regression model, which performed poorly across the board. The best model was a tie between the k-nearest-neighbors model and the bagging ensemble model, which both received an R2 score of .64 when run on all features. Though this is not a perfect score, it means the mean average error was just .2, which is small enough to be useful when optimizing program schedules and selling ad space.

1. Problem

In 2020, live television is in danger of being eclipsed by streaming video on demand. More and more people are refusing to pay for over-the-air broadcasting altogether, opting instead for streaming services such as Netflix or Disney+. In such a time and considering the ever-increasing cost of producing TV shows, it is important to know how well a show will do and how best to get the largest audience out of the shows you have.

One of the primary ways TV stations measure performance of over-the-air broadcasts is with the TV rating point system. Each rating point corresponds to 1% of the population watching television in a given area at any time, so if a particular airing of a show receives a 2.3 rating point score for Austin, out of all people watching live TV during that airing, 2.3% were watching that particular show. The data is compiled and calculated by the Nielsen Corporation and is then sold to TV stations and intermediaries.

Though rating is not the whole picture, and stations frequently look at additional statistics such as share, households, or social media engagement to gauge success, it is still one of the best single metrics to point to to say how well a show is doing. Beyond letting producers know if a show is still worth putting money into, rating is used for a variety of decisions, such as which shows to air in which time slots, and how much advertisers may be willing to pay. For a TV station, the importance of understanding how well shows do cannot be overstated.

2. Literature Review

2.1 Predicting TV Ratings

Despite the commercial importance of such research, there is little available in the literature about what techniques have been used to previously predict ratings, and next to nothing

published within the past decade. This lack of published information is likely because this data and research is proprietary – expensive to obtain from Nielsen and difficult to legally distribute. Additionally, given the competitive nature of the market, it's no surprise TV stations would want to keep such research to themselves.

However, some researchers have found ways around this, either by collecting the data themselves or by simply not publishing the data alongside the paper to avoid the limiting legal restrictions. Although it's not a perfect comparison, we can reference regression research done in fields outside of TV ratings to better understand techniques to apply to this problem.

A report by Nielsen (Sereday & Cui, February 2017) explored the idea of using machine learning techniques on ratings data. Unfortunately, the report is not technical and is aimed at a non-expert audience, so it spends most of its time explaining basic concepts like choosing features and cross validation. Although Nielsen didn't publish the data they used and didn't disclose the absolute performance of their models, they did explain their approach, which was simply to test a wide variety of models and see which predicted the ratings most accurately. In the end, a kind of ensemble regressor called a gradient boosting regressor performed the best relative to every other model they used according to a combination of metrics – weighted mean absolute percentage error, R-squared, and ease of implementation. The report is not concerned with the exact same problem as this thesis – Nielsen was focused more on comparing across channels and across stations, while my dataset is limited to a single channel from a single station. However, the kind of data they had access to is similar to mine – since my data comes primarily from Nielsen – so their findings and optimal model should be similar too.

Another report (Danaher, Dagger, & Smith, 2011) tried to model TV ratings, and was more forthcoming with the results. Using Nielsen-like data – plus a few added features like show genre, episode duration, whether it's a rerun, and and whether it's a holiday – and a Bayesian model (a variant of the common linear regression algorithm), they were able to achieve a .848 R2 score (for an explanation of R2 score see the "Metric choice" section below) across all 5 of their

channels. However, that does not tell the total picture – the predictability of each channel by itself varied immensely, from .832 to .414. This implies that predicting the performance of TV models is difficult – what may be a good result for one channel may be a mediocre result for another – and thus comparing results across studies performed on different datasets may be of limited usefulness.

There are a few more studies that try to model TV ratings ((Pagano, 2015) and (Danaher & Dagger, 2012), for example) published in the past decade that all reinforce the primary ideas discussed above. Each one finds that a different model works best for their own data, and their final predictive abilities vary immensely. Since there are so few recent studies on predicting TV ratings, and the results differ so much between studies and even between channels on the same dataset, it makes sense to reference regression studies done in other fields instead. Online or broadcast media are different in many ways – much more data is available on online users, and online media has a chance of going viral and reaching millions of views quickly out of seemingly nowhere, for example – but at the end of the day, they are still both media. Users find their favorite channels and shows and tend to watch them most of the time, and, through chance or word of mouth, more users stumble on certain channels or shows, and popularity grows. Meanwhile, as content grows stagnant or quality declines, users abandon channels and shows for others. This semi-competitive, consistent, and time-sensitive environment is the same for television, streaming services, and social media, and techniques used to model one field may find surprising success when applied to another.

2.2 Predicting Streaming and Web Content Popularity

A 2010 study (Szabo & Huberman, 2010) set out to predict the popularity of future content on the video-sharing site YouTube and a social networking site called Digg (where users submitted links to content on other sites similar to Twitter or Tumblr today). Using a relatively

simple linear model, they were able to achieve a relative error of 10% for Digg posts after 30 days, and a similar error for YouTube videos (after tracking their performance for ten days after being uploaded).

A 2014 Swedish thesis (Arvidsson, 2014) applied neural networks to predicting on-demand video ratings for a Netflix-like platform named Videoplaza. The prevailing method of predicting ratings on the platform was to use a seasonal averaging approach. For example, if you wanted to predict how well shows would do on a Wednesday, you would simply average all the ratings for shows on the past n Wednesdays, where n is an arbitrary integer. This is a simple approach, but despite using then-state-of-the-art feed-forward neural networking architecture, Arvidsson was unable to predict ratings more accurately on average than the seasonal average approach. This was likely because the viewership for the platform, much like over-the-air TV ratings, was highly "seasonal." Viewership always spiked in the evenings, and weekends tended to do better than weekdays.

Though this is not an encouraging result, implying either that ratings are trivially easy to predict or so difficult the difference in complexity between seasonal averaging and neural networks was negligible, there are useful conclusions to draw here. First, neural networks may not be the best way to predict TV ratings, for whatever reason, so other types of modeling methods are crucial. Second, Arvidsson's models did the best when trained on only the past week of airings, not all data available. This is likely because the data goes out of date quickly, and only the most recent airings impact how well the next airings are going to do. So it is also important to test training on smaller subsets of data to see if improvements can be gleaned that way.

A 2017 study (Trzciński & Rokita, 2017) similarly tried to predict the popularity of online videos from YouTube and Facebook, this time using a support vector regression model. Instead of solely relying on audience statistics and past user behavior, as previous studies had, they analyzed the videos themselves, extracting a variety of features such as color, faces, text, scene dynamics, and so on.

A survey of web prediction studies (Tater, 2014) found that the kind of model was not very important – dozens of studies used dozens of different models, but every study found positive results on their own datasets. Instead of the choice of model, the survey found that choice and extraction of features was an important and underexplored aspect of web predictive modeling.

These findings mirror my own experiences when looking at previous research into predicting TV ratings. Almost every study found that a different kind of model worked best for their data, and most studies used the same basic set of features provided by the raw Nielsen data. Moving forward with research in this area, it seems like the class of model used is less important than the data provided to the model and how that data is encoded.

2.3 Other Time Series Regression Studies

Though expanding our search to studies predicting streaming ratings and web popularity gives us more research to work with, there may still be useful information if we expand our search even further. Studies in other fields, such as medical analyses, may have more resources and history to build on. Though the data may be different, these papers are a good source to find more potentially useful models and ideas for feature engineering.

A survey of time series regression studies in environmental epidemiology (Bhaskaran, Gasparrini, Hajat, Smeeth, & Armstrong, August 2013) analyzed many different studies and condensed them into a general outline for time series analysis. Though there is lots of information that is only relevant to diagnosing patients – such as controlling for seasonality and long-term trends and allowing for delayed exposure effects – there are still useful conclusions to draw here. Interestingly, instead of opting for potentially more complex models such as neural networks, the researchers only considered various forms of Poisson regression models. These models are very similar to basic linear regression models, except that they assume the data follows a Poisson distribution, not a normal distribution. Additionally, they have more nuance than linear regression

algorithms, with more options to tune the algorithm to a specific dataset by manipulating the algorithm slightly.

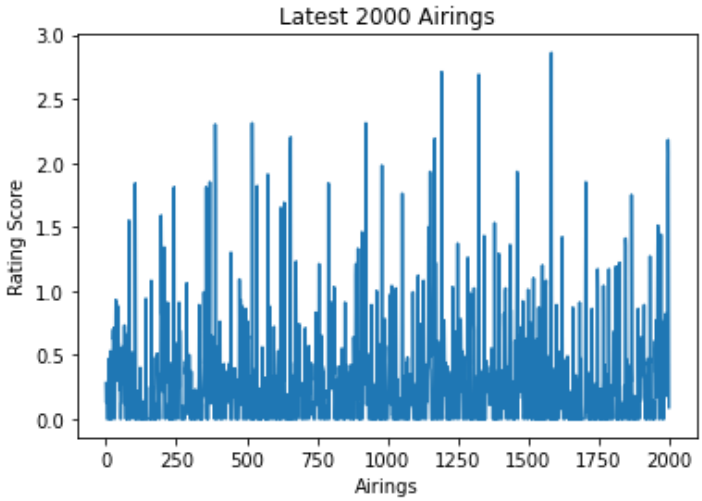
A 2019 study used neural networks to predict landslides in China (Yang, Yin, Lacasse, & Liu, 2019). Specifically, they used a kind of neural network called a long short-term memory (LSTM) network. An LSTM is a kind of recurrent neural network, which are specialized for time-series analysis by continually feeding old records as inputs to the network at each training step. Access to this data allows the network to recognize temporal patterns in data that regular networks can't. The researchers were able show that the LSTM model outperformed the older support vector machine model. This implies that neural networks, and specifically the recurrent LSTM variant, are worth looking at in addition to traditional data science models for time series data like mine.

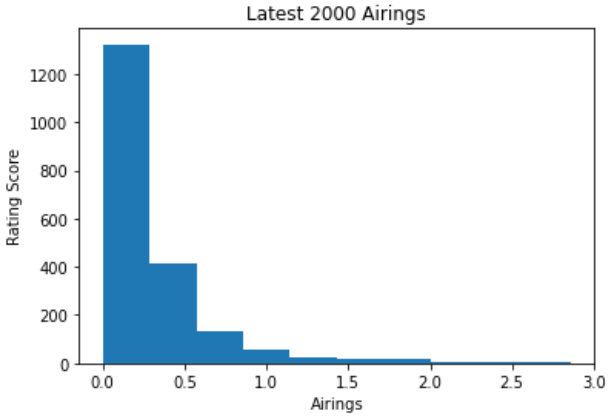
3. Data

My data is all airings on the public-television channel KLRU from the public broadcasting station KLRU-TV in Austin from September 2014 to September 2019. This includes 1,300 unique shows shown in homes across the Austin metro area. The shows cover all kinds of genres and feature local, national, and international content, from the KLRU-filmed music show *Austin City Limits* to Victorian-era BBC dramas like *Downton Abbey* to local news shows to PBS children's cartoons. Most airings are 30 minutes to an hour long, though on rare occasions some shows go multiple hours long. Though most shows have low ratings – of the 65,000 airings, 13,000 of them have a rating of zero – some shows spike as high as 11.

Date	Time	Title	Episode	Day	Daypart	HH
9/30/2014	5:00 AM	(string)	(integer)	Tue	Early Morning	(float)

An example of a record, scrubbed of Nielsen-owned data





3.1 Metric Choice

There are a variety of metrics available for scoring regression models, but in the end I decided to go with r-squared (R2) score, also known as the coefficient of determination, which measures how much of the variance of an output variable is explained by the input variables (Qian & Chen, 2013). In this case the input is the predictions of my model and the output is the actual ratings values, so R2 score measures the level or correlation between the two. The formal definition of the calculation is put below.

$$r = \frac{n(\Sigma xy) - (\Sigma x)(\Sigma y)}{\sqrt{[\, n\Sigma x^2 - (\Sigma x)^2 \,][\, n\Sigma y^2 - (\Sigma y)^2 \,]}}$$

A score of 1 indicates a perfect predictor, 0 means the predictor does as well as uniformly guessing the mean value, and anything negative means the predictor is worse than naively guessing the mean value and ignoring all features.

R2 was an attractive metric because it doesn't correspond to the magnitude of the output data, allowing me to easily see how good or bad the predictor is in a vacuum and not just relative to the other models. Normalizing data usually allows mean absolute error or mean squared error to have the same property, but the majority of my data is clustered near zero, even after normalizing, which skews those two error metrics to seem smaller than they really are. So in the end I went with R2 as my main metric.

R2 score is not the full picture – many problems are sensitive to the time it takes to train the models, for example, or care more about false negatives than false positives – but for the purposes of this initial exploration and this particular problem, it is by far the most important measure of the success of a model.

3.2 Nielsen Data Encoding

3.2.1 Time, Day of the Week, Daypart

Most of my data is categorical, but most models can't take in non-numeric data like strings. One of the most popular ways of converting categorical data to numeric is by one-hot-encoding it. This is where you create a column for each possible category in the

original column, and have a one if the original column has that value or a zero if it doesn't.

Genre		Genre_Music	Genre_Romance	Genre_Drama
Music		1	0	0
Romance		0	1	0
Romance		0	1	0
Music		1	0	0
Drama		0	0	1

The raw data is almost all categorical data that can be well encoded by a simple one-hot-encoding method. Of the six given features – date, time of day, title of the show, episode number, day of the week, and daypart (early morning, mid-morning, early afternoon, etc.) – time, day of the week, and daypart can all be encoded this way without creating unwieldy numbers of columns that will slow down and mislead the models.

3.2.2 Date

Date is always tricky to encode in time series data. Though there are multiple possibly useful features that can be extracted from it – month, day of the month, week, year, day of the year, etc. – it usually doesn't make much sense to include the raw date, and it definitely doesn't make sense to one-hot encode it. I decided to go with simply converting the date to the number of days from earliest airing in the dataset, preserving the continuity of the data and no added columns. Theoretically, this allows the model to group all the airings for a particular day together, but, considering the average rating didn't change much during this period, this may not be very significant information.

3.2.3 Episode Number

Episode number was straightforward. There should be little to no relationship between similar episode numbers of different shows – how well the 5th episode of *Austin City Limits* does has no impact on how well the 5th episode of *Antiques Roadshow* does. Going off this assumption, I normalized the episode number by show, so 0 is the earliest

episode of that show from the dataset and 1 is the most recent, regardless of which numbered episodes are aired in other shows.

3.2.4 Title

The last feature was the title of the show. There were two main kinds of encoding I considered for this – one-hot encoding and bag of words, which is essentially one-hot encoding every title on each individual word (including frequencies of each word in the original title), rather than the complete string. Because the title can include more information than just a hash-able label for the record (for example, most shows that are about the news include the word "news" in the title, and one-hot-encoding doesn't represent the relationship between the shows), the bag of words encoding should do a better job than the one-hot encoding, making titles with common words have similar values.

However, bag of words can falsely make titles seems similar just because they share a common word like "the." An alternative that fixes this problem is TF-IDF encoding, which stands for "term frequency – inverse document frequency." TF-IDF is like bag of words in that it creates a vector for each word that shows up in a title, with a zero if that word doesn't show up in the title for a given record, and something else if that word is present. In bag of words, that something else is simply a one, but TF-IDF has a number between 0 and 1 denotes the significance of the word if it is present. The term frequency for a given word and title is simply the number of times a word appears in the title divided by the total number of words in the title. The inverse document frequency for a given word is the log of the total number of titles divided by the number of titles with that word. TF-IDF is simply these two numbers multiplied together. In essence, TF-IDF determines how significant it is that a given word is present in a given title. It is high if it is a rare word that appears multiple times in a given title and low if it's a common word

that only appears once in the title, or not at all. Though this number is more difficult to interpret than the bag of words encoding, it is frequently more useful than just the raw bag of words encoding for models.

Because it's not obvious which encoding would be better, I decided to run the experiment with all three kinds of encoding – one hot encoding, bag of words, and TF-IDF – to see what fits this dataset the best.

3.3 Feature Engineering

One of the best ways to improve predictive models is with a technique called feature engineering. Most models are good at finding patterns in data, but no model is infallible, and certain models are better at finding certain patterns than others. If there is an important pattern in your data not explicitly expressed in a feature of its own, it may be worth it to give that pattern its own column.

For example, theoretically the date feature has the day of the week feature built into it. TV ratings data is heavily tied to the weekly cycle – people simply have more time to watch shows on the weekend than on a Tuesday, so having access to the day of the week is important. Though a model may not use the words "Monday," "Tuesday," "Wednesday," and so on, it may be able to find the seven-day pattern in the date. Because of this, and because the date can encode additional data like which shows aired on the same day, you would expect a model with access to just the date would do better than a model with access to just the day of the week.

However, for this data, that is not the case. Running linear regression with access to just the date of each airing gets a score of -.01 while running the same algorithm with just day of the week gets a score of .02. This is because the model is either not smart enough to find the pattern itself or finds the pattern but it is so obscured by noise that it doesn't get weighed as heavily. Giving the model explicit access to the day of the week boosts performance by making it much easier for the model to pick up on the weekend-weekday pattern.

3.3.1 Running Average

One of the two classes of features I engineered was a running average for a given combination of metrics. If the metric given is Title, then for each unique show aired I keep a running tally of how well each airing of that show has done so far, and add a cell denoting the average so far. If it's the first airing of a given show, it's running average is given a 0.

Title	Rating	Running_Average_Title
ACL	0.5	0
ACL	1	0.5
Masterpiece Classic	0.2	0
ACL	0.5	0.75
Masterpiece Classic	0.3	0.2

It works for multiple features as well – if both Daypart and Title are given, then I keep a running average for Title, as above, for Daypart, and for Title and Daypart at once, meaning keeping the average of how well a given show in a given Daypart does. This has an advantage over simply calculating the mean for each metric at once and making another pass to add the extra cells for each row – it doesn't ever contain information from the future. Row 1's running average isn't affected by how well Row 101 does, even if they share the same features, which more accurately models what information should be available to the model at the time of prediction. And of course, no results from the training set are used in calculating the running average – rows in the training set simply use the most recent running average value in the training set without changing it.

3.3.2 Previous Airing

The other class of features that I created, called "previous airing," used a similar idea, but simpler – instead of having a running average for a given combination of

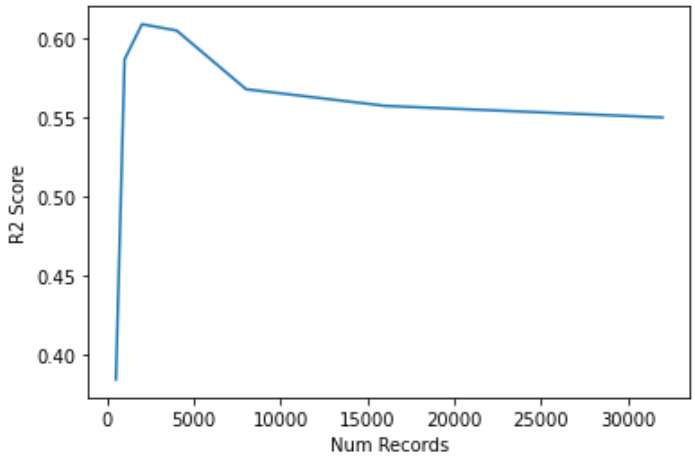
features, just use the latest value. If the first airing of *Austin City Limits* has a rating of .5, then the next airing of *ACL* (which had a rating of .8) will have its previous_title column be .5, then the next airing of ACL will have a previous_title value of .8, and so on.

This is all to more directly model the relationship between similar records and help the algorithm find more subtle patterns in the data. Theoretically, how well the latest airing of a given show did should be a good predictor of how the next airing will do since they are so close in time. This also avoids a potential downfall of running averages, which is that those might be misled by old, outdated data. How well an airing of a given show did last month should matter very little compared to how well the latest airing of that show did, so including previous airing in addition to the running average should shore up that potential hole in the data.

Title	Rating	**Prev_Airing_Title**
ACL	0.5	0
ACL	1	0.5
Masterpiece Classic	0.2	0
ACL	0.5	1
Masterpiece Classic	0.3	0.2

3.4 Data Size

One possible concern is that my dataset is not large enough to make meaningful predictions, or that the models I'm using require more records than I have access to. To make sure this isn't the case, I trained several kinds of models while limiting the number of records they had access to iteratively.



KNearestNeighbors on [Title, Time] features

As you can see, the best score came from using only the most recent 2,000 records, and performance actually drops slightly after that. TV ratings data goes out of date quickly, and how well a show performed a month ago has little bearing on how it will perform tomorrow, so it makes sense that removing meaningless records will keep the models from getting confused.

3.5 IMDb Data

Another concern about the Nielsen data was that it provided no information about the relationships between shows. Even though two shows are both Victorian English sitcoms, the Nielsen data alone won't give the model that information. IMDb, or the internet movie database, is a crowd-sourced website like Wikipedia that depends on users to submit information on movies and TV shows. Additionally, it allows its users to rate shows and movies on a 1 to 10 scale, which allows some insight into how positively or negatively a given piece of media has been received by the public. While, like Wikipedia, it is not completely reliable, it is nonetheless largely accurate and contains large amounts of information not found in any other one place.

Using this database, I annotated the existing Nielsen data with features that I thought were relevant. This included features like genres, time of filming, and how users on IMDb rated the show. IMDb had even more information, such as actors, directors, studio, and so on, but I was worried about creating tens of thousands of probably irrelevant features after one-hot-encoding

everything – greatly increasing the training time and possibly lowering the accuracy of many of my models – so I decided to exclude that data for this thesis.

While IMDb did not have information on all the shows in my database, it still held enough to annotate more than half of my records, and after examining my learning curves, I still had more than enough records to reach the max accuracy for my models, so I decided to simply exclude the shows that IMDb had no information on. Further research on this dataset could go back and manually add the information for each missing show, but that is out of the scope of this initial exploration.

tconst	titleType	primaryTitle	originalTitle	isAdult	startYear	endYear	runtimeMinutes	genres
tt0000001	short	Carmencita	Carmencita	0	1894	\N	1	Documentary, Short

The header from one of the three IMDb database files, title-basics

Code to annotate the Nielsen data file with the IMDb data

3.6 Cross Validation

An important part of all prediction models is cross-validation. Most of the models used rely heavily on random guesses that slowly converge to be meaningful. Sometimes, the model happens to make a specific guess that gives it a seemingly significant score when used on a particular training set, but when used on other training sets, it does poorly. This is a fairly rare occurrence, but when testing hyperparameters, you frequently end up calling a model a hundred times or more, and chances are at some point it's going to make a lucky guess that still has no predictive benefits.

To protect against this, most research uses k-fold cross validation, where the dataset is split into k segments each time a model is called with new hyperparameters. Each segment takes turns being the testing set used to measure the predictive ability of the model trained on the rest of the data, essentially eliminating the possibility that the model is just making a lucky guess rather than learning the trends in the training set.

However, this only works on data where the order of the records doesn't matter. For time-series data like mine, how well a show ten airings before does has a large impact on how well the latest airing does – the order matters and taking big chunks of consecutive data out messes with that temporal relationship. Additionally, metrics like running average and previous airing don't make much sense when the testing set isn't at the end – if you know the running average before and after an airing, it's pretty easy to guess how well that airing did.

I had to use something else to preserve the time series nature of my data and allow me to continue using the running average and previous airing features while still having the functionality of k-fold cross validation. The models performed best when running on a small subset of my dataset – only 2,000 records (with the first 1600 records as the training set and the last 400 as the testing set.) when there are 38,000 records in my total dataset. This allowed me to train the model k times on different subsets of the dataset, keeping the models from making lucky guesses while still allowing me to use my engineered features as long as I recalculated them just for that subset of data.

4. Approach

Because every previous TV ratings prediction study found a different model worked best for their data, I had no way of knowing what was best for my data. I decided to follow the approach of the Nielsen study (Sereday & Cui, February 2017), which was to implement many classes of models to find what was best suited to my particular dataset. The problem is that there are dozens of mainstream models used for time-series regression, and each one brings its own challenges. Most have a handful of hyperparameters (like how many neighbors to compare to in the k-nearest-neighbors regressor) that have a massive impact on predictive ability, and I couldn't rule out any kind of model until I tried many combinations of hyperparameters.

The sklearn library made this easy. It includes more than a dozen diverse, commonly used regression models already implemented, with good documentation detailing what hyperparameters are the most important to experiment with. It's hard to know the right range of hyperparameter values to evaluate ahead of time – the number of neighbors in the nearest neighbors algorithm, for example, is highly problem dependent. Instead, it is preferable to start with large ranges with large step values (going up fifty neighbors every iteration), and then narrow it down for successive iterations.

However, the sklearn library is not without its limitations – specifically neural networks. There is only one kind of neural network implemented by the library – a multilayer perceptron (MLP) – but some of the latest advances in regression modeling come from more advanced, recurrent neural networks.

Implementing the hyperparameter tuning for the keras models, however, was significantly more difficult than the sklearn models. The keras library functions by manually stacking different combinations of hidden layers, rather than relying on a prebuilt optimized implementation like sklearn models with simple options to feed in as arguments. This does give users much more control over the algorithm, allowing them to fit it to their problem better, but the exponentially branching combinations of layers and parameters make exhaustive tests of every possible combination difficult. Because the keras library also lacks any automatic way of testing a variety of hyperparameters, such as number of hidden layers, batch size, time steps, and so on, I implemented my own. The difference between different combinations of hyperparameters is massive, so trying many combinations, either automatically or manually, is necessary.

There are many more options for keras networks than sklearn models, and the formulation gets significantly more complicated.

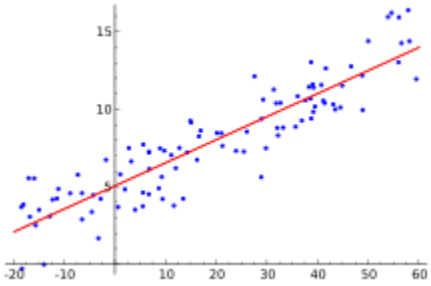
Every neural net has an input layer – where the data comes in – and an output layer – where the prediction comes out. There are usually many hidden layers between these outer layers to help the network model complex patterns, but more layers don't always lead to better results, so thorough testing on the number of layers is important. You also have to test different configurations of neurons – or the number of nodes in each layer for the next layer to connect to. Sometimes the number of neurons

decreases steadily as the network nears the output layer; other times it stays constant until the output. Keras also offers a kind of layer called dropout layers to put between regular hidden layers. These are layers that remove some arbitrary amount of connections between layers, usually 20%. Though it may seem counterintuitive to remove connections, it forces the remaining neurons to be more efficient with finding patterns in the data that can be applied to new datasets. Without these dropout layers, there is the possibility that the network is simply memorizing the training dataset, which would be useless for predicting future ratings.

Additional hyperparameters for keras networks are batch size, number of epochs, and time steps (only for the LSTM network). Batch size is how many a network trains on at a time. For example, a batch size of 100 means you use 100 records at a time to train the network at each step. Once the network is done training on those 100 and updates its internal weights, it takes in the next 100 records, and so on. Batch sizes can be used to categorize your data, such as all the airings in a particular day, so that the model gets trained on similar data at each step. The number of epochs is the number of times you repeat this process. At each epoch, you train on all the available data in batch sized segments. Having multiple epochs is important to let the model fully learn the dataset and give it enough time to update all of its weights, but if the model is allowed to train for too long it will over-train and simply memorize the training data, and lose predictive accuracy.

5. Models

5.1 Linear Regression



https://en.wikipedia.org/wiki/Linear_regression#/media/File:Linear_regression.svg

Linear regression is the most simple and naïve regression algorithm, short of simply returning the average of the inputs. It simply tries to find the straight line – or hyperplane in the case of high dimension data – that best represents the data, typically calculated by minimizing the sum of the squares of the distance each point has from the line (Freedman, 2009). This works accurately and efficiently for linear data, but data in the real world is rarely truly linear.

5.2 Decision Trees

Decision trees are relatively simple models. They form a tree where each split is a split along a particular feature. So if the first split is on the day of the week, you may have the left be all records that occurred on Mondays, and on the right all the rest. Then you split both of those datasets, and so on. Then, when you want to make a prediction based on the tree, you just follow the record down the tree according to the split criteria until you reach a leaf node, which could either be a single record or a group of records, and take the average rating of those (Park & Ko, 2005).

Decision trees are good for datasets that have a lot of binary features since the binary tree data structure naturally fits the splits in the data. Each time a feature is one-hot-encoded it

essentially creates several (or sometimes hundreds) of binary features, so considering how most of my features are one-hot-encoded, it would make sense that a decision tree would do well.

5.3 Graphical Models

5.3.1 K Nearest Neighbors

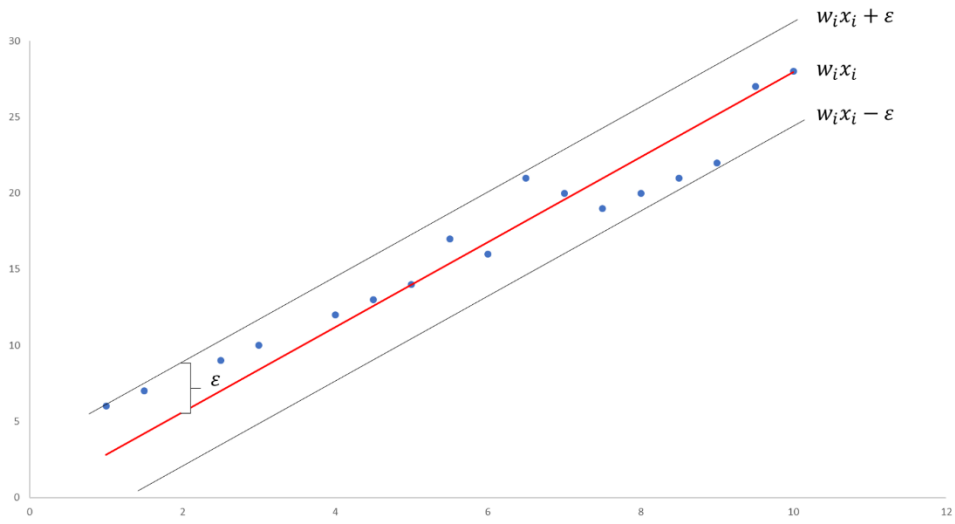
The k nearest neighbors algorithm works by comparing a test point to k other points that are closest, then returning the average of the independent variable – the rating score in this case – among those neighbors (Lee, 2019). Because that distance is calculated as the sum of the differences between the two points along each dimension, properly normalized data is crucial. If one feature has a range of 0 to 1, and another feature has a range of 0 to 100, the second feature is going to dominate the first, and the algorithm may incorrectly dismiss the first feature as unimportant. However, if everything is normalized properly, it can be a surprisingly effective algorithm, especially for strongly clustered data.

In TV, what tends to dominate how well an airing will do is when it's aired. There are orders of magnitude more people watching TV at primetime than at three in the morning, so no matter how good a show is, if it isn't aired at the right time, almost no one will watch it. Because the data is so strongly clustered around time of airing, I expected the nearest neighbors algorithm to do well, though it may be unable to find more subtle patterns in the data such as trend differences in episode number, or extract meaning from the running average and previous airing features.

5.3.2 SVM

Regression SVMs are similar to linear regression algorithms in that they try to find a hyperplane that best represents the data, but they are much more sophisticated.

Instead of punishing all errors from points that don't lie directly on the hyperplane, they use slack to define the acceptable range of points to be in. Essentially, in two-dimensional space, instead of defining a line that minimizes least squares of all points, you try to find the tube of space of a certain radius that fits the most points while minimizing the error of points that are out of that tube (Lee, 2019).



SVM visualization courtesy of towardsdatascience.com

5.4 Ensemble Models

5.4.1 Bagging Ensemble

The bagging ensemble regressor is the first ensemble model I considered. Ensemble models are a class of models of models. They're based on the idea that if you have a bunch of inaccurate – but better than just guessing the mean – models that are all independent and diverse, then averaging the predictions of each of the models will get you much closer to the true result than any of the individual models would get you by themselves (Zhou, 2012). The only problem is that ensemble methods take orders of magnitude longer to train than the base regressors they average together.

Bagging ensembles work by training many versions of the same base model such as a decision tree or k-nearest neighbors regressor. By training each model independently

on a random subset of features and records, they guarantee the independent and diverse requirement for ensemble methods to get better results than their base models. For my base models, I decided to use the best tuned versions of several of my other models – K Nearest Neighbors, SVM, and sklearn MLP.

5.4.2 Boosting Ensemble

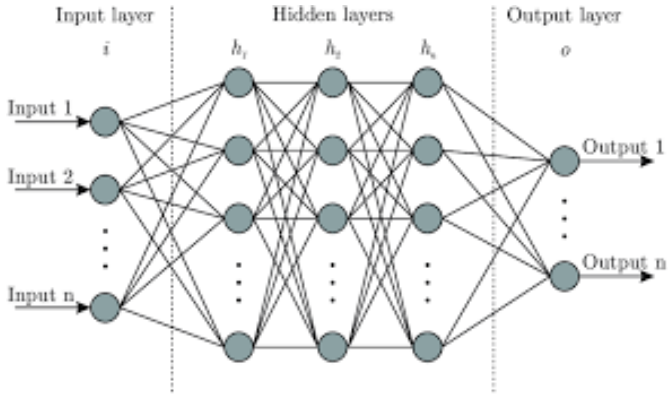
Boosting ensemble models work a bit differently than bagging ensembles. Instead of training many models independently, then averaging them together, boosting algorithms train the models iteratively, where the previous model influences the next (Zhou, 2012). Essentially, younger models learn from the mistakes of the older models. I'm trying two different implementations of boosting ensemble models – gradient boost and XGBoost.

Gradient boosting is a version of the general boosting algorithm which distinguishes itself by using loss functions to identify the shortcomings of the older models, and usually does better than traditional boosting methods. Using stochastic gradient descent, they are able to find the arrangements of weights to features that results in the least error.

XGBoost, on the other hand, maximizes training speed and flexibility. There are dozens of potential hyperparameters that alter the algorithm significantly, from the base model to boost to the basic form of regression each model uses. This allows XGBoost to model linear, logistic, and Poisson-distributed data very well.

5.5 Neural Networks

5.5.1 MLP



https://www.researchgate.net/figure/Artificial-neural-network-architecture-ANN-i-h-1-h-2-h-n-o_fig1_321259051

Neural networks have revolutionized the regression modeling field in the past decade. Though they don't work better on every dataset, they are especially good at finding non-linear correlations between features in data compared to traditional data science models. This allows them to find more subtle relationships between features, ones that the researcher may not even know about, and patterns that less advanced algorithms would have found only through extensive feature engineering (Ghosh, De, & Pal, 2007).

Neural networks can do this by having several hidden layers between the input layers (features) and output layers (prediction). By tuning the weights between nodes of different layers iteratively, they are able to effectively represent the relationships between the data. The way different neural network algorithms distinguish themselves from each other is how they connect the nodes of all these different layers. The most basic kind of neural network, the MLP, is also called a fully connected neural network because it has connections between every pair of nodes in subsequent layers. Though this is the most obvious approach, it has resulted in highly accurate predictions across a variety of problems, frequently beating out more complicated algorithms.

5.5.2 LSTM

A more complex version of the neural network is the recurrent neural network (RNN). Instead of having the linear setup of MLPs, where one layers smoothly leads to another in a line, RNNs feature loops, where some of the output of deeper layers gets passed back to the input of shallower layers. Essentially, a RNN's input is not just the current record, but also every record that have already been exposed to. This is called back-propagation and is the key that makes recurrent neural networks so good for analyzing sequential data. You could scramble the order of feed-forward networks and they would return the same result, but RNNs are able to take meaning from particular sequences of data. In addition to the two-dimensional input data, they create a third, temporal dimension, so that more recent records are weighted more heavily than old records.

This makes them very well suited for natural language processing, but it comes with a downside – frequently, RNNs end up dismissing old data almost completely, and only rely on the most recent handful of records. In some cases, this is preferred behavior, but most problems benefit from the extra context older data offers. For example, if a RNN is parsing a paragraph, by the time it reaches the end it may have left out important information from the first sentence, which obviously can result in warped results.

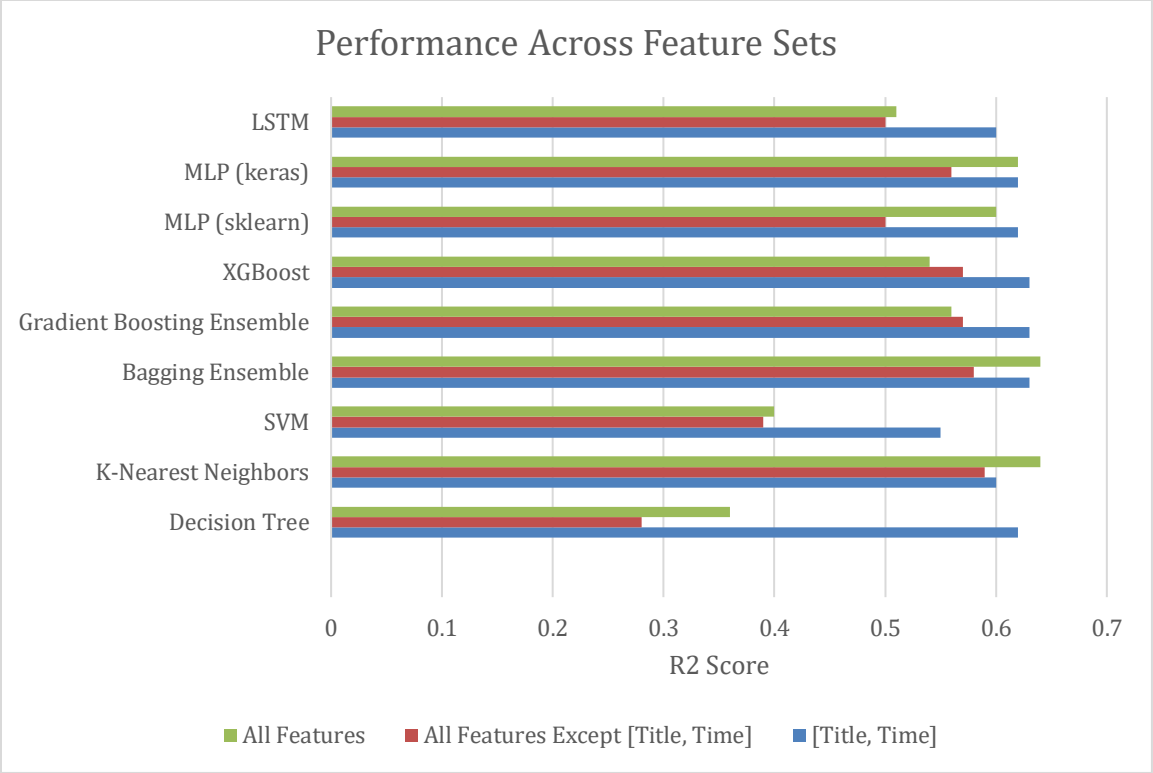
Long short-term memory (LSTM) RNNs are a solution to this problem. LSTM layers have memory cells that remember old, but still relevant data and pass it on to the next layer as inputs. By doing this, LSTM networks are able to make better use of longer datasets than typical recurrent neural networks, and are some of the best models of time-series data in recent years (Greff, Srivastava, Koutník, Steunebrink, & Schmidhuber, 2017).

6. Results

6.1 Models and Hyperparameters

Every dataset appears to have its own limit to how predictable it is. One of the studies discussed above (Danaher, Dagger, & Smith, 2011) used the same techniques to predict TV ratings across four different channels, and got widely different results for each one, implying that some kinds of TV channels and programs are more predictable than others. This by itself is not a surprise – certain programs, like Monday night football, for example, have consistently high ratings, while other programs have large discrepancies between airings. Because different channels host different shows, it's expected that certain channels would be more stable. What was a surprise was the magnitude of the difference – one channel had more than twice the R2 score as the other using the same techniques. It's impossible to know exactly where a given channel lies on that predictability range, and that is part of the reason predicting TV ratings is so difficult.

With all that said, the limit of the predictability for my dataset seems to be in the .6-.65 R2 score range, at least with these features. I ran all the models with just title and time of day as features, and the best models got an R2 score of .63. I ran them again with all features except time and time of day and got a nearly equivalent score of .59. If both feature sets are equivalently good at predicting the rating, it would be expected that combining them would significantly increase the predictive capabilities of the models. However, actually running the models on all features still resulted in an essentially equivalent score of .64, implying that giving either feature set more information doesn't improve performance. It is possible there is more information out there that can inform the models better than my current data.



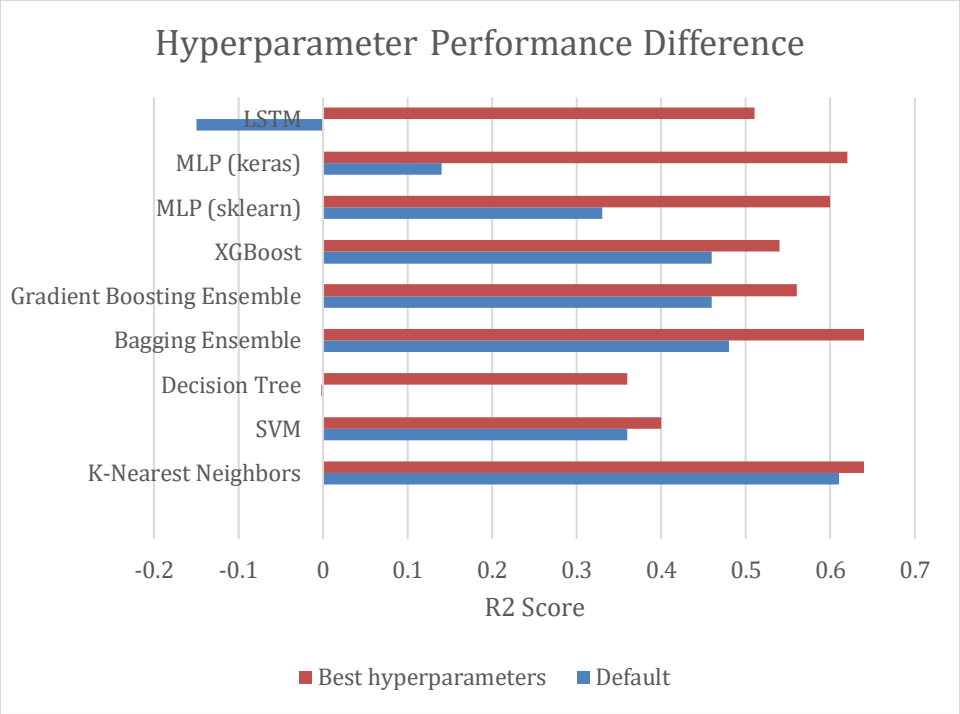
Model	[Title, Time]	All Features Except [Title, Time]	All Features
Linear Regression	-112224636.4	-20546.58	-2846540.15
Decision Tree	0.62	0.28	0.36
K-Nearest Neighbors	0.6	0.59	0.64
SVM	0.55	0.39	0.4
Bagging Ensemble	0.63	0.58	0.64
Gradient Boosting Ensemble	0.63	0.57	0.56
XGBoost	0.63	0.57	0.54

MLP (*sklearn*)	0.62	0.5	0.6
MLP (keras)	0.62	0.56	0.62
LSTM	0.6	0.5	0.51

All numbers denote R2 score. Many of these models employ some random sampling techniques, so even after cross-validation the numbers may shift as much as +-.03 between trials.

Some of the models scored worse with more features, but that is expected – several of these models are subject to the "curse of dimensionality" where their predictive ability worsens on higher dimension datasets. However, every model did better with just title and time than everything but title and time. Even though the running average idea is so similar to the seasonal average techniques used by Arvidsson, which outperformed his neural networks, on KLRU's data it still is not able to compete with these two basic Nielsen features. All that matters is what is on and at what time. Some models were able to extract modest benefits from having access to all the features compared to just title and time, but the greatest improvement was a mere .04 difference. This may imply that predicting on-demand video ratings and predicting over-the-air TV ratings may be more different problems than I originally assumed, but it is dangerous to read too much into just two data points, and more research in the area would be required to make that claim.

Instead of feature engineering or model choice, like most data mining problems, what I found worked best to improve my predictive capacity was hyperparameter tuning. The difference between default and correctly tuned hyperparamatized models could be huge, as much going from -.15 to .51, and with the right parameters, even initially weak models were able to do reasonably well.



Model	Default	Best hyperparameters
Linear Regression	-2846540.15	(No hyperparameters)
Decision Tree	-0.002	0.36
K-Nearest Neighbors	0.61	0.64
SVM	0.36	0.4
Bagging Ensemble	0.48	0.64
Gradient Boosting Ensemble	0.46	0.56
XGBoost	0.50	0.56
MLP (sklearn)	0.33	0.6

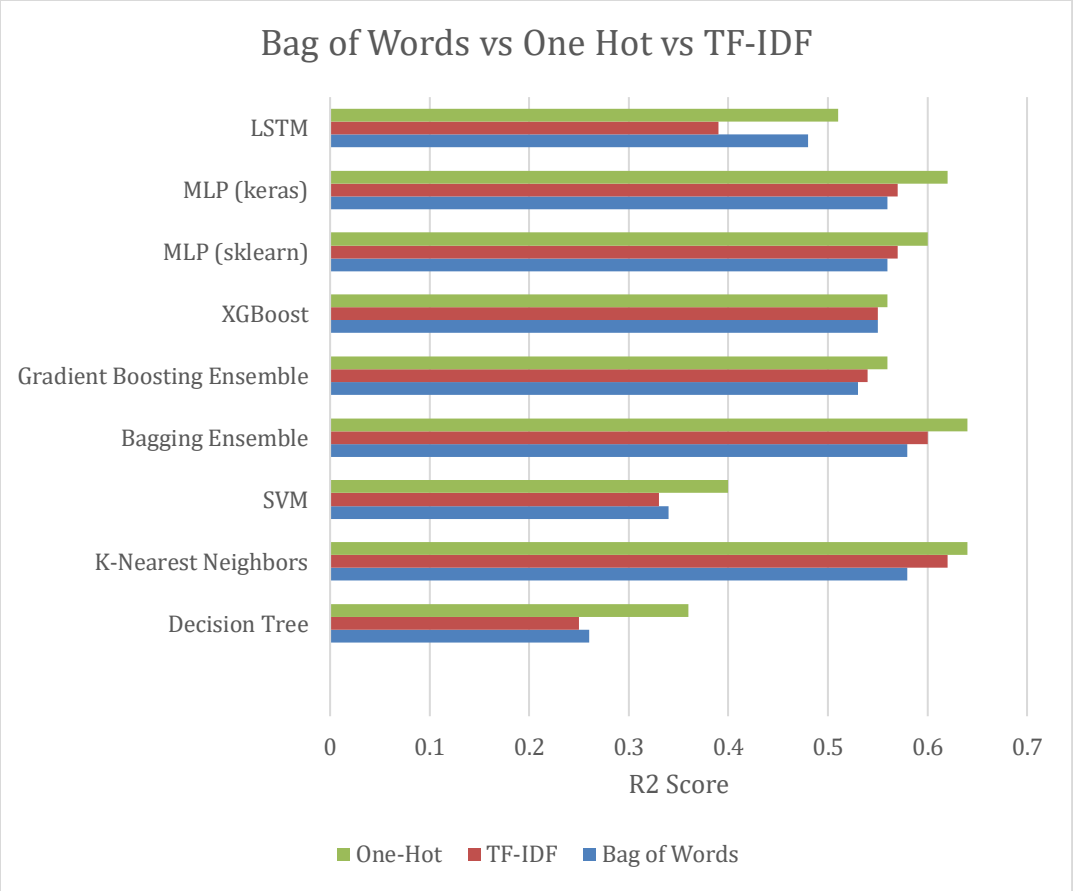
MLP (keras)	0.14	0.62
LSTM	-0.15	0.51

Run on all features. Since the keras models didn't have a default configuration, instead I implemented a simple model for each kind of network with only one hidden layer and relatively few neurons.

6.2 One-Hot Encoding vs Bag of Words vs TF-IDF

All of the above results with Title use the one-hot-encoding method, not the bag-of-words or TF-IDF methods, because the latter two did worse than the one hot encoding for all models. Though the bag of words and TF-IDF encodings are theoretically would be able to tie together programs with similar titles – like two news shows that both have the word "news" in their title, for example – there are so few instances of that in the data that it just confuses the models.

Most titles offer only limited insight into the content and type of show that they're attached to – the title *Austin City Limits*, for example, doesn't imply any sort of musical content unless you have prior knowledge of the show. In these cases, all the bag of words algorithm does is weaken the connection between airings of the same show by polluting it with random connections to other shows that happen to have the word "The" in the title. Though TF-IDF is supposed to help make common words like that insignificant, that kind of encoding tends to break down on short "documents" or titles in my case. Considering most titles are three words or less, TF-IDF is simply not as useful as it is on other kinds of datasets.



Model	Bag of Words	TF-IDF	One-Hot
Linear Regression	-3172843.52	-538695.89	-2846540.15
Decision Tree	0.26	0.25	0.36
K-Nearest Neighbors	0.58	0.62	0.64
SVM	0.34	0.33	0.4
Bagging Ensemble	0.58	0.6	0.64

Gradient Boosting Ensemble	0.53	0.54	0.56
XGBoost	0.55	0.55	0.56
MLP (sklearn)	0.56	0.57	0.6
MLP (keras)	0.57	0.57	0.62
LSTM	0.48	0.39	0.51

Run on all features

7. Future Work

Predicting TV ratings is not an easy problem. The patterns in the data are not very difficult to see, even for humans – ratings spike in the evenings and on weekends and fall between those times. Rather, what makes it difficult is the pseudo-random behavior of humans and how seemingly unrelated events can have large effects on the rating score.

There are thousands of factors that determine whether someone watches TV at a given time and thousands more that determine what they decide to watch once they sit down. If there is some important event going on, like a natural disaster in a neighboring city, the news programs will have better ratings. If someone's friend just recommended the latest season of *Downton Abbey*, that person is more likely to tune into *Downton Abbey* that night. If someone comes from a family that's really into sports, that person is more likely to watch sports, and so on. Without knowing everyone's TV preferences and everything that happened earlier that day that might influence their decision, it's impossible to make a perfectly accurate TV ratings predictor. Though some of these can be measured in aggregate, such as genre preferences in a population, human behavior is so complex it might as well be random from an algorithmic perspective, and large numbers of people can suddenly change their mind to tune into another program or do something else with their afternoon at any given time.

Future work in this area should work towards finding and incorporating data that may help to explain and reduce this randomness. If a show is trending on social media that day, it will likely see a large boost in ratings. If another show in the same timeslot on a different channel is airing its finale, ratings will likely go down. If there is a large event in town like the South by Southwest film festival in Austin, fewer people will be home watching TV, so ratings across the board may drop, and so on. Each one of these data points is tricky to model. Social media popularity is ephemeral and can't be predicted more than a day or two out with much accuracy, so that data will constantly have to be refreshed and the models retrained before any new predictions can be made. Modeling competition from other channels is difficult because it requires so much domain knowledge and familiarity with the TV landscape of a particular area – an algorithm won't be able to tell you if another TV show's premier will cut into your ratings or not because there's no previous data for that show to train off of. Of course, guesses can be made based on genre, amount of advertising, star power, and so on, but that data can be difficult to get ahead of time. Modeling how the attendance of local events will affect your ratings can also be difficult thanks to the scarcity of available data – usually only a few big events happen each year, but if the data only goes back two years, that doesn't give enough information to extrapolate a meaningful pattern from.

Beyond predicting the ratings of existing shows, there is the even more interesting and difficult problem of predicting the ratings of a completely new show. TV stations pour millions of dollars into every new show, and there's no way to know if their investment will be worth it until it starts airing. Executives currently try to predict the success of a show on a variety of metrics – the past successes of the people on the project, the success of the pilot episode, the quality of the writing, and so on – but almost all of these are completely qualitative and rely on the executive's intuition. Being able to reliably predict the success of new shows in a quantitative way may help stations avoid millions in wasted investments.

The issue is that this is a very hard problem – orders of magnitude harder than merely predicting the ratings of shows that have already aired. When there is not past evidence of the performance of a show, you have to rely on other data to feed into the models. This could include the full cast and working staff for the show, along with the average success of every show every person on that cast has worked on

in the past. You could look at the advertising budget, or the popularity of teasers on social media, even feed the script of the show itself into the model.

Each one of these possibilities, both in general ratings prediction and specifically predicting new shows, warrant a study of their own to possibly explain and demystify the randomness behind TV ratings. And, of course, technological improvements mean that more advanced kinds of predictive models will be coming out every year. The models in this paper were selected based on their historical results from the past decade, but new and experimental models may be the key to making strides in this field. This problem space is deep and rich with mineable information, and, using modern techniques and innovative feature engineering, we can make great improvements in predicting even seemingly unstable and unpredictable datasets.

8. Conclusion

What my studies have found is that there appears to be a hard limit on how predictable my dataset is. Though two independent feature sets have similar performance with the same models, combining the feature sets results in no net improvement. Despite extensive feature engineering, data reduction, and hyperparameter tuning, no combination of model and feature set was able to go above a .64 R2 score. The two models that did the best were the k-nearest neighbors and the bagging ensemble regressors.

This is not necessarily a bad result. Previous studies on predicting TV ratings had wildly varying successes in how well they were able to predict the rating score, and some even have large differences in predictability across channels in the same study. Additionally, this information is definitely not useless to TV stations. An R2 score of .64 translates to a .2 average error from the actual rating, which is small enough that this line of research could be an invaluable tool for assisting the experts who create the daily schedule for the station, and may suggest untapped areas of potential to boost ratings.

However, this paper alone is not enough to prove that my findings are the limit of this dataset. Though I've based my study off the crucial findings of the papers in the literature review, there are still

possibilities that have not been fully studied yet. Combining the strategies in this paper with additional data from other areas, such as social media performance, the ratings of all the other major stations in the area, or even the daily trends of video streaming, could all potentially better inform the models. However, TV ratings remain unstable and susceptible to rising or falling for seemingly random reasons. TV shows are art, and art is notoriously difficult to evaluate, and until researchers are able to encode the quality of the writing of a given episode or how well a particular actor performed that day, it will still be impossible to fully predict how well a show will do.

9. References

ALTAŞ, D., & ÖZTUNÇ, H. (2013). An Empirical Analysis of Television Commercial Ratings in Alternative Competitive Environments Using Multinomial Logit Model. *Eurasian Journal of Business and Economics*, 39-51.

Arvidsson, J. (2014). *Forecasting on-demand video viewership ratings using neural networks.*

Bhaskaran, K., Gasparrini, A., Hajat, S., Smeeth, L., & Armstrong, B. (August 2013). Time series regression studies in environmental epidemiology. *International Journal of Epidemiology*, 1187-1195.

Danaher, P. J., Dagger, T. S., & Smith, M. S. (2011). *Forecasting television ratings.* Carlton, Australia: International Journal of Forecasting.

Danaher, P., & Dagger, T. (2012). Using a nested logit model to forecast television ratings. *International Journal of Forecasting*, 607-622.

Freedman, D. A. (2009). *Statistical Models: Theory And Practice.* Cambridge: Cambridge University Press.

Ghosh, A., De, R. K., & Pal, S. K. (2007). *Pattern Recognition and Machine Intelligence.* Kolkata: Second International Conference, PReMI 2007.

Greff, K., Srivastava, R. K., Koutník, J., Steunebrink, B. R., & Schmidhuber, J. (2017). LSTM: A Search Space Odyssey. *IEEE*, 2222-2232.

Lee, W.-M. (2019). *Python Machine Learning.* Indianapolis: John Wiley & Sons, Inc.

Pagano, R. e. (2015). *Prediction of TV ratings with dynamic models.* Milan, Italy.

Park, J., & Ko, H. (2005). Effective acoustic model clustering via decision-tree with supervised learning. *Speech Communication*, 1-13.

Sereday, S., & Cui, J. (February 2017). Using Machine Learning to Predict Future TV Ratings. *Nielsen Journal of Measurement*.

Szabo, G., & Huberman, B. A. (2010). Predicting the popularity of online content. *Communications of the ACM*, 80-88.

Tater, A. e. (2014). A survey on predicting the popularity of web content. *Journal of Internet Services and Applications*.

Trzciński, T., & Rokita, P. (2017). Predicting Popularity of Online Videos

Qian, Q., & Chen, S. (2013). Co-metric: a metric learning algorithm for data with multiple views. *Frontiers of Computer Science*, 259-269.

Yang, B., Yin, K., Lacasse, S., & Liu, Z. (2019). Time series analysis and long short-term memory neural network to predict landslide displacement. *Landslides*, 677-694.

Zhou, Z.-H. (2012). *Ensemble Methods: Foundations and Algorithms.* Boca Raton: Taylor and Francis Group.

10. Source Code

```
import pandas as pd
import numpy as np

def build_dictionaries():
    title_akas = pd.read_csv(FOLDER_PATH + 'title-akas.tsv', sep='\t')
    title_basics = pd.read_csv(FOLDER_PATH + 'title-basics.tsv', sep='\t')
    title_ratings = pd.read_csv(FOLDER_PATH + 'title-ratings.tsv', sep='\t')
    dataset = pd.read_csv(FOLDER_PATH + 'Input.csv')

    id_to_title = {}
    titles = np.unique(dataset['Title'].values)
    for index, row in title_akas.iterrows():
        title = row['title']
        if title in titles:
            id_to_title[row['titleId']] = title

    title_to_genres = {}
    title_to_year = {}
    for index, row in title_basics.iterrows():
        if row['tconst'] in id_to_title:
            title = id_to_title[row['tconst']]
            title_to_genres[title] = row['genres'].split(',')
            title_to_year[title] = row['startYear']
    title_to_ratings = {}
    title_to_votes = {}
    for index, row in title_ratings.iterrows():
        if row['tconst'] in id_to_title:
```

```
            title = id_to_title[row['tconst']]
            title_to_ratings[title] = row['averageRating']
            title_to_votes[title] = row['numVotes']
    return title_to_genres, title_to_year, title_to_ratings, title_to_votes

def annotate_data():
      # Get imdb data and annotate the input dataset
      title_to_genres, title_to_year, title_to_ratings, title_to_votes = bui
      ld_dictionaries()
      genres_list = []
      years_list = []
      ratings_list = []
      votes_list = []
      dataset = pd.read_csv(FOLDER_PATH + 'Input.csv')
      for index, row in dataset.iterrows():
          title = row['Title']
          if title in title_to_genres and title_to_genres[title] != '\\N':
              genres_list.append(title_to_genres[title])
          else:
              genres_list.append(pd.NaT)
          if title in title_to_year and title_to_year[title] != '\\N':
              years_list.append(title_to_year[title])
          else:
              years_list.append(pd.NaT)
          if title in title_to_ratings and title_to_ratings[title] != '\\N':
              ratings_list.append(title_to_ratings[title])
          else:
              ratings_list.append(pd.NaT)
          if title in title_to_votes and title_to_votes[title] != '\\N':
              votes_list.append(title_to_votes[title])
          else:
              votes_list.append(pd.NaT)

      dataset['Genres'] = genres_list
      dataset['Year'] = years_list
      dataset['Rating'] = ratings_list
      dataset['NumVotes'] = votes_list
      dataset = dataset.dropna()

      dataset['Year'] = pd.to_numeric(dataset['Year'])
      dataset['Rating'] = pd.to_numeric(dataset['Rating'])
      dataset['NumVotes'] = pd.to_numeric(dataset['NumVotes'])

      # save to csv
      dataset.to_csv(FOLDER_PATH + 'ProcessedData.csv')
```

```
import pandas as pd
import numpy as np
from sklearn import preprocessing
from itertools import combinations
from sklearn.feature_extraction.text import TfidfVectorizer, CountVector
izer

FOLDER_PATH = '/content/drive/My Drive/Thesis/'
```

```
TRAIN_TEST_RATIO = .8
FILE_PATH = FOLDER_PATH + 'ProcessedData.csv'

def calc_running_average(dataset, given_features):
    # Mapping of a set of values of features to a tuple (running sum, co
unt)
    # Ex: ('Music', 'Prime Time', 'Tuesday): (5.8, 4)
    features_to_average = {}
    feature_indices = []
    for feature_name in given_features:
        feature_indices.append(dataset.columns.tolist().index(feature_na
me))
    rating_index = dataset.columns.tolist().index('HH Rating')

    # List to return and add to the dataset
    running_average = []
    for index, row in dataset.iterrows():
        features_vals = []
        for i in feature_indices:
            features_vals.append(row[i])
        features_vals = tuple(features_vals)
        if index < (len(dataset) * TRAIN_TEST_RATIO):
            rating = row[rating_index]
            if features_vals in features_to_average:
                current_sum = features_to_average[features_vals][0]
                current_count = features_to_average[features_vals][1]
                running_average.append(current_sum / current_count)
                features_to_average[features_vals] = (current_sum + rati
ng, current_count + 1)
            else:
                running_average.append(0.0)
                features_to_average[features_vals] = (float(rating), 1)
        else:
            # Ensure no data leakage by not including ratings from the t
est set
            # in the running average, just use most recent value from tr
aining set
            if features_vals in features_to_average:
                  current_sum = features_to_average[features_vals][0]
                  current_count = features_to_average[features_vals][1]
                  running_average.append(current_sum / current_count)
            else:
                running_average.append(0.0)
    return running_average
```

```
def calc_prev_airing(dataset, given_features):
    # Mapping of a set of values of features to the rating of the most
    # recent occurence of those values
    features_to_prev = {}
    feature_indices = []
    for feature_name in given_features:
        feature_indices.append(dataset.columns.tolist().index(feature_na
me))
    rating_index = dataset.columns.tolist().index('HH Rating')

    # List to return and add to the dataset
    prev_airing = []
    for index, row in dataset.iterrows():
        features_vals = []
        for i in feature_indices:
            features_vals.append(row[i])
        features_vals = tuple(features_vals)
        if index < (len(dataset) * TRAIN_TEST_RATIO):
            rating = row[rating_index]
            if features_vals in features_to_prev:
                prev_airing.append(features_to_prev[features_vals])
            else:
                prev_airing.append(0.0)
            features_to_prev[features_vals] = rating
        else:
            # Ensure no data leakage by not including ratings from the t
est set
            # in the running average, just use most recent value from tr
aining set
            if features_vals in features_to_prev:
                  prev_airing.append(features_to_prev[features_vals])
            else:
                prev_airing.append(0.0)
    return prev_airing

def one_hot_encode(df, column_name):
    df = pd.concat([df, pd.get_dummies(df[column_name], prefix=column_na
me)], axis=1)

    df.drop([column_name], axis=1, inplace=True)
    return df


def move_col_to_end(df, column_name):
    col = df[column_name]
```

```
    df = df.drop([column_name], axis=1)
    df = pd.concat([df, col], axis=1)
    return df


def normalize_col(column):
    return (column - column.min()) / (column.max() - column.min())


def standardize_col(column):
    x = column.values.astype(float)
    scaler = preprocessing.StandardScaler()
    x_scaled = scaler.fit_transform(x)
    return pd.DataFrame(x_scaled)


def remove_outliers(dataset, column_name):
    quant = dataset[column_name].quantile(0.999)
    return dataset.where(dataset[column_name] < quant)


def bag_of_words_encode(dataset, col_name):
    list_of_words = set()
    for cell in dataset[col_name]:
        for token in cell.split():
            list_of_words.add(token)

    new_df = pd.DataFrame(0, index=np.arange(len(dataset[col_name])), co
lumns=list_of_words)
    index = 0
    for cell in dataset[col_name]:
        for token in cell.split():
            new_df.loc[index, token] = new_df.loc[index, token] + 1
        index += 1
    dataset = pd.concat([dataset, new_df], sort=True, axis=1)
    dataset = dataset.drop(columns=[col_name])
    return dataset


def tf_idf(dataset, col_name):
    column = dataset[col_name].values.astype('U')
    vectorizer = TfidfVectorizer()
    vectors = vectorizer.fit_transform(column)
    new_df = pd.DataFrame(vectors.todense().tolist(), columns=vectorizer
.get_feature_names())
    dataset = pd.concat([dataset, new_df], sort=True, axis=1)
```

```
    dataset = dataset.drop(columns=[col_name])
    return dataset

def one_hot_list_feature(dataset, col_name):
    list_of_features = set()
    for cell in dataset[col_name]:
        for feature in cell.split(','):
            list_of_features.add(feature)

    new_df = pd.DataFrame(0, index=np.arange(len(dataset[col_name])), co
lumns=list_of_features)
    index = 0
    for cell in dataset[col_name]:
        for feature in cell.split(','):
            new_df.loc[index, feature] = new_df.loc[index, feature] + 1
        index += 1
    dataset = pd.concat([dataset, new_df], sort=True, axis=1)
    dataset.drop([col_name], axis=1, inplace=True)
    return dataset


def all_permutations(list):
    result = []
    for x in range(0, len(list)):
        for y in combinations(list, x):
            result.append(y)
    final_element = []
    for x in list:
        final_element.append(x)
    result.append(final_element)
    return result


def load_dataset_features(features, running_average_features=None, prev_
airing_features=None, num_records='all', offset=0,
                          bag_of_words=False, tf_idf=False):
    dataset = pd.read_csv(FILE_PATH, header=0)
    dataset = dataset.loc[:dataset.shape[0] - offset, :]

    if not num_records == 'all':
        dataset = dataset.loc[(dataset.shape[0] - num_records):, :]
        dataset = dataset.reset_index(drop=True)

    dataset['HH Rating'] = normalize_col(dataset['HH Rating'])
```

```
    cols_to_drop = dataset.columns
    cols_to_drop = cols_to_drop.drop('HH Rating')
    for feature in features:
        if feature in cols_to_drop:
            cols_to_drop = cols_to_drop.drop(feature)
        if feature in ['Year', 'Rating', 'NumVotes']:
            dataset[feature] = normalize_col(dataset[feature])
    if running_average_features != None:
        for feature_list in all_permutations(running_average_features):
            name = 'running_average'
            for feature_name in feature_list:
                name += '_' + feature_name
            dataset[name] = calc_running_average(dataset, feature_list)
    if prev_airing_features != None:
        for feature_list in all_permutations(prev_airing_features):
            name = 'prev_airing'
            for feature_name in feature_list:
                name += '_' + feature_name
            dataset[name] = calc_prev_airing(dataset, feature_list)
    if 'Date' in features:
        dates = []
        dates_col = []
        i = 0
        for index, row in dataset.iterrows():
            if row['Date'] not in dates:
                i += 1
            dates_col.append(i)
        max_val = max(dates_col)
        for index in range(0, len(dates_col)):
            dates_col[index] = float(dates_col[index])/max_val
        dataset['Date'] = dates_col
    if 'Day' in features:
        dataset = one_hot_encode(dataset, 'Day')
    if 'Time' in features:
        dataset = one_hot_encode(dataset, 'Time')
    if 'Daypart' in features:
        dataset = one_hot_encode(dataset, 'Daypart')
    if 'Episode' in features:
        dataset['Episode'] = pd.to_numeric(dataset['Episode'], errors='c
oerce')
        dataset = dataset.dropna()
        title_to_ep_list = {}
        for index, row in dataset.iterrows():
            title = row['Title']
            ep = row['Episode']
```

```
            if title in title_to_ep_list:
                if ep not in title_to_ep_list[title]:
                    title_to_ep_list[title].append(ep)
            else:
                title_to_ep_list[title] = [ep]

        for index, row in dataset.iterrows():
            title = row['Title']
            if title in title_to_ep_list:
                ep_list = title_to_ep_list[title]
                if (max(ep_list) - min(ep_list)) == 0:
                    new_ep = 0
                else:
                    new_ep = (row['Episode'] - min(ep_list)) / (max(ep_l
ist) - min(ep_list))
                dataset.loc[[index], ['Episode']] = new_ep
            else:
                dataset.loc[[index], ['Episode']] = 0
    if 'Title' in features:
        if bag_of_words == True:
            dataset = bag_of_words_encode(dataset, 'Title')
        else if tf_idf == True:
            dataset = tf_idf(dataset, 'Title')
        else:
            dataset = one_hot_encode(dataset, 'Title')
    if 'Year' in features:
        dataset['Year'] = normalize_col(dataset['Year'])
    if 'Genres' in features:
        dataset = one_hot_encode(dataset, 'Genres')
    if 'Rating' in features:
        dataset['Rating'] = normalize_col(dataset['Rating'])
    if 'NumVotes' in features:
        dataset['NumVotes'] = normalize_col(dataset['NumVotes'])

    dataset = dataset.drop(columns=cols_to_drop)
    dataset = move_col_to_end(dataset, 'HH Rating')
    # dataset = remove_outliers(dataset, 'HH Rating')
    dataset = dataset.dropna()
    # convert to np array
    dataset = dataset.values
    dataset = dataset.astype('float32')

    return dataset

def train_test_split(dataset):
```

```
    train_size = int(len(dataset) * TRAIN_TEST_RATIO)
    train, test = dataset[0:train_size, :], dataset[train_size:len(datas
et), :]
    num_columns = dataset.shape[1]

    # split into input (X) and output (y) variables
    X = train[:, :num_columns - 1]
    y = train[:, num_columns - 1]
    X_test = test[:, :num_columns - 1]
    y_test = test[:, num_columns - 1]
    return X, y, X_test, y_test


def train_test_split_3d(dataset, time_steps):
    train_size = int(len(dataset) * TRAIN_TEST_RATIO)
    train, test = dataset[0:train_size, :], dataset[train_size:len(datas
et), :]

    num_rows = train.shape[0]
    num_columns = train.shape[1]
    num_samples = int(num_rows / time_steps)
    try:
      train = train.reshape(num_samples, time_steps, num_columns)
    except:
      return (None, None, None, None)

    X = train[:, :, :num_columns - 1]
    y = train[:, :, num_columns - 1]

    test_num_rows = test.shape[0]
    test_num_columns = test.shape[1]
    num_samples = int(test_num_rows / time_steps)
    try:
      test = test.reshape(num_samples, time_steps, test_num_columns)
    except:
      return (None, None, None, None)
    X_test = test[:, :, :test_num_columns - 1]
    y_test = test[:, :, test_num_columns - 1]

    return X, y, X_test, y_test
```

```
from sklearn.linear_model import LinearRegression
from sklearn.neighbors import KNeighborsRegressor
from sklearn.svm import SVR
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import BaggingRegressor, GradientBoostingRegressor
from sklearn.neural_network import MLPRegressor
```

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, LSTM
from sklearn.metrics import r2_score
from xgboost import XGBRegressor

def evaluate_model(model):
    sum_score = 0
    for trial in range(0, 5):
        dataset = get_dataset(trial)
        X, y, X_test, y_test = train_test_split(dataset)
        model.fit(X, y)
        score = r2_score(y_test, model.predict(X_test))
        sum_score += score
    return sum_score / 5

def linear_regression():
    model = LinearRegression()

    score = evaluate_model(model)
    print('Linear regression', score)
    return score

def nearest_neighbors():
    best_score = None
    best_nn = None
    best_weights = None
    best_algorithm = None
    for num_neighbors in range(15, 25, 2):
        for weights in ['uniform', 'distance']:
            for algorithm in ['auto', 'ball_tree', 'kd_tree', 'brute']:
                model = KNeighborsRegressor(n_neighbors=num_neighbors, weigh
ts=weights, algorithm=algorithm)
                score = evaluate_model(model)
                if best_score == None or score > best_score:
                    best_score = score
                    best_nn = num_neighbors
                    best_weights = weights
                    best_algorithm = algorithm
                if VERBOSE:
                    print(score, num_neighbors, weights)
    print('KNN', best_score, 'num neighbors', best_nn, 'weights', best_weigh
ts, 'algorithm', best_algorithm)
    return best_score

def svm():
    best_score = None
    best_kernel = None
    best_eps = None
    for kernel in ['linear', 'poly', 'rbf', 'sigmoid']:
        for epsilon in [.001, .01, .05, .15, .25]:
            model = SVR(kernel=kernel, epsilon=epsilon)
            score = evaluate_model(model)
            if VERBOSE:
                print(score, kernel, epsilon)
            if best_score == None or score > best_score:
                best_score = score
                best_kernel = kernel
```

```
                best_eps = epsilon
    print('SVM', best_score, 'kernel', best_kernel, 'epsilon', best_eps)
    return best_score


def decision_tree():
    best_score = None
    best_splitter = None
    best_depth = None
    best_features = None
    for splitter in ['best', 'random']:
        for max_depth in list(range(200, 400, 50)) + [None]:
            for max_features in [None, 'sqrt', 'log2', .1, .2]:
                model = DecisionTreeRegressor(splitter=splitter, max_depth=m
ax_depth, max_features=max_features)
                score = evaluate_model(model)
                if best_score == None or score > best_score:
                    best_score = score
                    best_splitter = splitter
                    best_depth = max_depth
                    best_features = max_features
                if VERBOSE:
                    print(score, splitter, max_depth, max_features)
    print('Decision Tree', best_score, 'splitter', splitter, 'max depth', be
st_depth, 'max features', best_features)
    return best_score

def bagging_ensemble():
    best_score = None
    best_est = None
    best_n = None
    best_boot = None
    best_features = None
    for base_estimator in [KNeighborsRegressor(n_neighbors=19, weights='dist
ance', algorithm='brute'),
                           SVR(kernel='linear', epsilon=.05),
                           DecisionTreeRegressor(splitter='random', max_dept
h=150, max_features=.2),
                           MLPRegressor(activation='logistic', solver='adam'
, learning_rate='constant', hidden_layer_sizes=(20,), batch_size=40)]:
        for n_estimators in range(3, 10):
            for bootstrap in [True, False]:
                for max_features in [.2, .5, 1]:
                    model = BaggingRegressor(base_estimator=base_estimator,
n_estimators=n_estimators, bootstrap=bootstrap, max_features=max_features)
                    score = evaluate_model(model)
                    if best_score == None or score > best_score:
                        best_score = score
                        best_est = base_estimator
                        best_n = n_estimators
                        best_boot = bootstrap
                        best_features = max_features
                    if VERBOSE:
                        print(score, base_estimator, n_estimators, bootstrap
, max_features)
```

```
    print('Bagging Ensemble', best_score, 'best estimator', best_est, 'best
n_estimators', best_n, 'bootstrap', best_boot, 'number of features', best_fe
atures)
    return best_score

def sk_mlp():
    best_score = None
    best_act = None
    best_solver = None
    best_lr = None
    best_n = None
    best_batch = None
    for activation in ['identity', 'logistic',  'tanh', 'relu']:
        for solver in ['lbfgs', 'sgd', 'adam']:
            for learning_rate in ['constant', 'invscaling', 'adaptive']:
                for n_layers in range(10, 40, 20):
                    for batch_size in range(1, 30, 10):
                        model = MLPRegressor(activation=activation, solver=s
olver, learning_rate=learning_rate, hidden_layer_sizes=(n_layers,), batch_si
ze=batch_size)
                        score = evaluate_model(model)
                        if best_score == None or score > best_score:
                            best_score = score
                            best_act = activation
                            best_solver = solver
                            best_lr = learning_rate
                            best_n = n_layers
                            best_batch = batch_size
                        if VERBOSE:
                            print(score, activation, solver, learning_rate,
n_layers, batch_size)
    print('SKLearn MLP', best_score, 'activation', best_act, 'solver', best_
solver, 'learning_rate_schedule', best_lr, 'best number of hidden layers', b
est_n, 'batch size', best_batch)
    return best_score

def gradient_boost():
    best_score = None
    best_loss = None
    best_n = None
    best_sub = None
    for loss in ['ls', 'lad', 'huber']:
        for n_estimators in list(range(50, 100, 10)):
            for subsample in [1.0, .1, .5, .9]:
                model = GradientBoostingRegressor(loss=loss, n_estimators=n_
estimators, subsample=subsample)
                score = evaluate_model(model)
                if best_score == None or score > best_score:
                    best_score = score
                    best_loss = loss
                    best_n = n_estimators
                    best_sub = subsample
                if VERBOSE:
                    print(score, loss, n_estimators, subsample)
    print('Gradient Boost', best_score, 'loss', best_loss, 'n estimators', b
est_n, 'subsampling', best_sub)
    return best_score
```

```
def keras_mlp_model(dataset, num_layers, batch_size, dropout, const_size, nu
m_epochs, dropout_val=.2, activation='relu'):
    X, y, X_test, y_test = train_test_split(dataset)

    model = Sequential()
    if not dropout and not const_size:
      model.add(Dense(2 ** num_layers, activation=activation, input_dim=X.sh
ape[1]))
      for size in range(num_layers - 1, -1, -1):
          model.add(Dense(2 ** size, activation=activation))
    if not dropout and const_size:
      model.add(Dense(dataset.shape[1], activation=activation, input_dim=X.s
hape[1]))
      for size in range(num_layers - 1, 0, -1):
          model.add(Dense(dataset.shape[1], activation=activation))
      model.add(Dense(1))
    if dropout and not const_size:
      model.add(Dense(2 ** num_layers, activation=activation, input_dim=X.sh
ape[1]))
      for size in range(num_layers - 1, -1, -1):
          model.add(Dropout(dropout_val))
          model.add(Dense(2 ** size, activation=activation))
    else:
      model.add(Dense(dataset.shape[1], activation=activation, input_dim=X.s
hape[1]))
      for size in range(num_layers - 1, 0, -1):
          model.add(Dropout(dropout_val))
          model.add(Dense(dataset.shape[1], activation=activation))
      model.add(Dropout(dropout_val))
      model.add(Dense(1))

    model.compile(loss='mse', optimizer='rmsprop')
    model.fit(X, y, epochs=num_epochs, batch_size=batch_size, verbose=0)
    return r2_score(y_test, model.predict(X_test))

def keras_mlp():
  best_num_layers = 0
  best_batch_size = 0
  best_dropout = 0
  best_const_size = 0
  best_num_epochs = 0
  best_score = None
  for num_layers in range(2, 6):
    for batch_size in range(40, 100, 20):
      for dropout in range(0, 2):
        for const_size in range(0, 2):
          for num_epochs in range(5, 25, 10):
            sum_score = 0
            for trial in range(0, 5):
                dataset = get_dataset(trial)
                val = keras_mlp_model(dataset, num_layers, batch_size, dropo
ut, const_size, num_epochs)
                sum_score += val
                if VERBOSE:
```

```
                    print(val, num_layers, batch_size, dropout, const_size, nu
m_epochs)
            final_score = sum_score / 5
            if best_score == None or final_score > best_score:
              best_num_layers = num_layers
              best_batch_size = batch_size
              best_dropout = dropout
              best_const_size = const_size
              best_num_epochs = num_epochs
              best_score = final_score
  print('Keras MLP', best_score, best_num_layers, best_batch_size, best_drop
out, best_const_size, best_num_epochs)
  return best_score


def get_shared_factors(number1, number2, max):
    result = []
    for x in range(1, max):
        if (number1 % x == 0) and (number2 % x == 0):
            result.append(x)
    return result

def keras_lstm_model(dataset, time_steps, batch_size, num_layers, dropout, c
onst_size, num_epochs, dropout_size=.2):
      X, y, X_test, y_test = train_test_split_3d(dataset, time_steps)
      if X is None:
        return -1

      model = Sequential()

      if dropout and const_size:
        model.add(LSTM(dataset.shape[1] * 2, return_sequences=True, batch_in
put_shape=(batch_size, time_steps, len(X[0, 0, :]))))
        for i in range(0, num_layers):
          model.add(Dropout(dropout_size))
          model.add(LSTM(dataset.shape[1] * 2, return_sequences=True))
        model.add(Dropout(dropout_size))
        model.add(Dense(y.shape[1]))

      if dropout and not const_size:
        model.add(LSTM(y.shape[1] * 2 ** num_layers, return_sequences=True,
batch_input_shape=(batch_size, time_steps, len(X[0, 0, :]))))
        for i in range(num_layers, 0, -1):
          model.add(Dropout(dropout_size))
          model.add(LSTM(y.shape[1] * 2 ** i, return_sequences=True))
        model.add(Dropout(dropout_size))
        model.add(Dense(y.shape[1]))

      if not dropout and const_size:
        model.add(LSTM(y.shape[1] * 2, return_sequences=True, batch_input_sh
ape=(batch_size, time_steps, len(X[0, 0, :]))))
        for i in range(0, num_layers - 1):
          model.add(LSTM(dataset.shape[1] * 2, return_sequences=True))
        model.add(LSTM(dataset.shape[1] * 2))
        model.add(Dense(y.shape[1]))

      else:
```

```
        model.add(LSTM(y.shape[1] * 2 ** (num_layers + 1), return_sequences=
True, batch_input_shape=(batch_size, time_steps, len(X[0, 0, :])))))
        for i in range(num_layers, 1, -1):
          model.add(LSTM(y.shape[1] * 2 ** i, return_sequences=True))
        model.add(LSTM(y.shape[1] * 2))
        model.add(Dense(y.shape[1]))
      model.compile(loss='mse', optimizer='adam')

      try:
        model.fit(X, y, epochs=num_epochs, batch_size=batch_size, validation
_split=.2, verbose=0)
        return r2_score(y_test, model.predict(X_test))
      except:
        return -1

def keras_lstm():
  best_time_step = 0
  best_batch_size = 0
  best_num_layers = 0
  best_dropout = 0
  best_const_size = 0
  best_num_epochs = 0
  best_r2 = None
  factors = get_shared_factors(2000, 400, 400)
  step_size = int(len(factors) / 3)
  if step_size == 0:
    step_size = 1
  for index in range(0, len(factors), step_size):
      for batch_size in range(1, 7):
          for num_layers in range(1, 8, 2):
              for dropout in range(0, 2):
                  for const_size in range(0, 2):
                      for num_epochs in range(5, 25, 10):
                          sum_score = 0
                          for trial in range(0, 5):
                              dataset = get_dataset(trial)
                              r2 = keras_lstm_model(dataset, factors[index],
 batch_size, num_layers, dropout, const_size, num_epochs)
                              sum_score += r2
                          final_score = sum_score / 5
                          if VERBOSE:
                              print(final_score, factors[index], batch_size,
 num_layers, dropout, const_size, num_epochs)
                          if best_r2 == None or final_score > best_r2:
                              best_r2 = final_score
                              best_batch_size = batch_size
                              best_time_step = factors[index]
                              best_num_layers = num_layers
                              best_dropout = dropout
                              best_const_size = const_size
                              best_num_epochs = num_epochs
  print('Keras LSTM', best_r2, best_time_step, best_batch_size, best_num_lay
ers, best_dropout, best_const_size, best_num_epochs)
  return best_r2

def xgboost():
    best_score = None
```

```
    best_booster = None
    best_n = None
    best_sub = None
    best_depth = None
    best_objective = None
    for booster in ['gbtree','gblinear','dart']:
        for n_estimators in list(range(50, 150, 90)):
            for subsample in [1.0, .1, .5]:
                for max_depth in range(1, 11, 3):
                    for objective in ['reg:squarederror', 'reg:logistic', 'c
ount:poisson']:
                        model = XGBRegressor(booster=booster, n_estimators=n
_estimators, subsample=subsample, max_depth=max_depth, objective=objective)
                        score = evaluate_model(model)
                        if best_score == None or score > best_score:
                            best_score = score
                            best_booster = booster
                            best_n = n_estimators
                            best_sub = subsample
                            best_depth = max_depth
                            best_objective = objective
                        if VERBOSE:
                            print(score, booster, n_estimators, subsample, m
ax_depth, objective)
    print('xgboost', best_score, 'booster', best_booster, 'n estimators', be
st_n, 'subsampling', best_sub, 'max depth', best_depth, 'objective', best_ob
jective)
    return best_score
```